IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

UTILITY PATENT APPLICATION FOR:

METHOD AND APPARATUS FOR EXCHANGING THE CONTENTS OF REGISTERS

Inventors:

Kevin D. Safford
4100 Suncrest Drive
Fort Collins, CO 80525

Patrick Knebel
2201 Greenmont Court
Fort Collins, CO 80524

HP Docket No. 10992268

I. FIELD

The present invention relates to digital computer systems, and more particularly, but not by way of limitation, to methods and apparatus for executing instructions in such systems.

II. BACKGROUND

In x86 computer systems, the floating point unit (FPU) comprises a plurality of data registers. Floating point instructions treat this plurality of data registers as a register stack. All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in a stack TOP field. Thus, load operations decrement TOP by one and load a value into the new top-of-stack register, while store operations store the value from the current top-of-stack register in memory and then increment TOP by one.

Many floating point instructions, however, only operate on the top one or two registers of a register stack. Thus, if the desired information is located in, e.g., the fourth stack register, one or more operations must be performed before the information in the fourth stack register can be moved into the top register of the stack where it can be operated upon. This creates a "bottleneck" in the stack. To this end, the floating point exchange register contents instruction (FXCH) is used in the IA-32 computer architecture to exchange the floating point information in a selected stack register with that in the top register of the stack. For example, the instruction

FXCH ST(0), ST(i)

will exchange the information in the top register in the stack (denoted ST(0)) with the *i*th register in the stack (denoted ST(i)). In this way, the bottleneck in the register stack can be alleviated by putting desired information at the top of the stack, where it can then be operated upon by most floating point instructions. More information regarding the FXCH instruction may be found in the *Intel Architecture Software Developer's Manual, Volumes 1-3*, which are hereby incorporated by reference.

In many computer architectures, instructions, such as the FXCH, must be executed by emulation because the native hardware that supports such an instruction is not present. One way of emulating the FXCH instruction in such architectures is through a technique called register renaming. In register renaming, the physical registers in question (e.g., ST(0) and ST(i)) are mapped into a stack register map. To exchange the contents of the two physical registers, the pointers that map the physical registers into the stack register map are changed or "re-pointed" from their original register to the other register, and thus the operation is performed. But, at least one problem with register renaming is that it requires that the pointers be stored in additional hardware which adds to the cost and complexity of the system as well as consuming valuable space.

Another way of emulating the FXCH is to sequentially execute at least three micro-code instructions as follows:

```
move temp := ST(0);
move ST(0) := ST(i);
move ST(i) := temp;
```

This is the traditional method of exchanging the contents of the register. This sequence of instructions uses a temporary register to switch the contents of the top register ST(0) and the *i*th register ST(i). This method of emulation, with its three micro-code instructions, consumes three times as many clock cycles as the single FXCH instruction and, in some cases, may consume even more, depending upon the latency associated with the move operations. Thus, there exists a need for methods and apparatus for emulating the FXCH instruction without adding excess hardware and that consumes relatively few clock cycles. More generally, there exists a need for methods and apparatus for exchanging the contents of two registers in a relatively quick and efficient manner.

III. SUMMARY

In one embodiment of the present invention, there is a processor based computer system having dependency checking logic and a register stack, wherein the system overrides the dependency logic such that move instructions associated with the stack registers may be executed in parallel. In another embodiment, the system operates such that it can be determined whether a stack underflow exception has occurred and if it has, the move instructions can be flushed, and a micro-code handler algorithm invoked that operates to allow execution of the move instructions in parallel without a stack underflow exception.

IV. BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a computer system including the present invention.

Figure 2 is block diagram of the processor of Figure 1.

Figure 3 is an illustration of pipelined or lockstep operations.

Figures 4 is a flow chart of the operation of the stack underflow fault micro-code handler of the present invention.

V. DETAILED DESCRIPTION

A. Description of an Embodiment

Figure 1 illustrates a computer system 10 in which the present invention may be implemented. The computer system 10 comprises at least one processor 20, main memory 30, and various interconnecting data, address, and control busses (numbered collectively as 40). An instruction set 50 (which may be a guest instruction set) and an operating system 60 may be stored in main memory 30. As illustrated in Figure 2, the processor 20 comprises a floating point unit 70, dispatch or dependency checking logic 80, at least two execution units 90a and 90b, micro-code ROM 100, a register stack 120 (in this embodiment the register stack 120 comprises eight individual registers 120(0)-(7)), a floating point tag word (FPTW) register 130, and various busses and interconnections (numbered collectively as 110). (One skilled in the relevant art will note that there need not be a separate floating point unit — the execution units are equally capable of executing floating point instructions).

Instructions are provided to the processor 20 from main memory 30. The instructions provided to the processor 20 are macro-code instructions that map to one or more micro-code instructions 140 stored in the micro-code ROM 100. The micro-code instructions can be directly executed by processor 20. Also stored in the micro-code ROM 100 are a set of micro-code handlers 150 that may be invoked to handle processor exceptions.

The floating point unit 70 accesses the register stack 120 to store and retrieve data in response to instructions. The FPTW register 130 is updated accordingly.

The processor 20 may have a pipelined architecture and may have allow for parallel processing of certain instructions. Dependency checking logic 80 operates to determine which instructions can be operated in parallel, i.e., whether to issue two instructions or one instruction per cycle to the execution units 90.

B. Method of Operation

1. Parallel Execution of Move Instructions

Assume that the processor 20 is presented with the following instructions:

ST(i): = move ST(0);
ST(0): = move ST(i);

In a sequential microprocessor, this sequence of code is not capable of exchanging the contents of the two registers (ST(0) and ST(i). In a sequential microprocessor, each instruction is executed independently and in serial fashion. Thus, in this sequence the first instruction will overwrite ST(i) before the second instruction reads ST(i), and the result placed in ST(0) will be ST(0) rather than ST(i). This is obviously an incorrect result. As noted, the traditional method for correctly performing the desired exchange would require an additional temporary location and an extra instruction.

However, in the present invention, the computer system 10 emulates the FXCH instruction by forcing the two move instructions to be executed in parallel (illustrated conceptually below):

ST(i): = move ST(0);   ST(0): = move ST(i);

Because these two instructions each modify a register used by the other instruction, hardware would normally inhibit them from being executed in parallel. Thus, in the present invention, a mechanism is provided that forces the hardware to override the dependency checking logic, and forces the hardware to execute the two instructions in parallel. Both instructions are provided to respective execution units 90a and 90b that (1) read their respective operand at substantially the same time, (2) write their results at substantially the same time (at least one clock cycle after the read), and (3) finish executing their instructions at substantially the same time (i.e., they operate in lockstep). The latency associated with the respective execution units 90a and 90b is such that this can be accomplished. This lockstep execution is illustrated conceptually in Figure 3. In this manner, the computer system 10 can effectively emulate the FXCH instruction in substantially less time than it would take to execute the three move instructions of a sequential system.

2. Stack Underflow Exception

In modern superscalar microprocessors (ones with the ability to execute multiple instructions in parallel) exceptions are precise, meaning that when an operation faults, all of the "younger" operations in the pipeline must be flushed. However, in the present invention, a temporary register is not used to hold intermediate results of operations in the pipeline -- i.e., the two instructions are completed in lockstep. Thus, if either one of the two move instructions above causes an exception, both of the move instructions are flushed and re-executed in parallel to prevent data corruption. However, that is not conventional operation for many microprocessors. That type of operation will work properly if the "first" instruction causes the exception, but not if the "second" instruction does -- in that case, conventional operation dictates that the "first" instruction would not be flushed (see example below):

ST(i) := mov ST(0)	ST(0) := mov ST(i)
"first"	"second"
"older"	"younger"

Thus, in the present invention, the computer system 10 flushes both operations when either of them causes a fault.

An exception that may occur when these move instructions are executed is called a "stack underflow" exception. A stack underflow exception occurs when an operation attempts to read the contents of an empty stack register 120(0)-(7). A floating point tag word stored in the FPTW register 130 indicates whether a stack register 120(0)-(7) is empty or not. A defined architectural response to a stack underflow is to replace the empty register with a QNaN, mark it as non-empty, and the perform the instruction again. While it is possible to add hardware inside the execution units 90a and 90b to indicate which of the two move instruction caused the stack underflow fault, that is not desired because of the additional complexity and cost.

Thus, in the computer system 10, a micro-code handler algorithm 200, such as that illustrated in Figure 4, is invoked by processor 20 when a stack underflow exception occurs. At block 210 of Figure 4, a stack underflow exception has occurred when an attempt to execute the two move operations in parallel was attempted. At block 220, the micro-code handler algorithm 200 causes the FPTW bits in the FPTW register 130 that correspond to the ST(0) register to be checked and then at block 230, a decision is made as to whether register ST(0) is empty or not. If register ST(0) is empty, at block 240 its contents are replaced with a QNaN and the corresponding FPTW bit is set to indicate that the ST(0) register is no longer empty. Proceeding now to block 260, emulation of the FXCH instruction is performed again (by issuing the two instructions in parallel again). If the register ST(0) was not empty, the exception must have occurred because register ST(i) was empty and, at block 250 the register ST(i) contents are replaced with a QnaN, the corresponding FPTW bit is set, and the emulation is performed again at block 260. At block 270, if a stack underflow exception occurs once again, it is known that both registers involved in the operation must have been empty originally and that this time, the ST(i) register caused the exception. The registers contents at this stage are:

ST(0) := QNaN ST(i) := empty

Accordingly, at block 280, the ST(i) register is loaded with a QNaN and at block 290, the emulation proceeds again, this time without any exceptions -- the QNaNs in the two registers will be harmlessly exchanged.

C. Remarks

By "forcing" the two move instructions to execute in parallel, emulation of the FXCH instruction may be achieved in substantially less time than previous methods and without adding hardware to the computer system 10. Furthermore, a microcode handler ensures correct execution of the emulated FXCH in the event of a stack underflow exception.

It will be readily apparent to those skilled in the art that innumerable variations, modifications, applications, and extensions of these embodiments and principles can be made without departing from the principles and spirit of the invention. For example, the methods and apparatus described herein may be applied to emulation of other instructions and to the handling of exceptions that occur when they are executed.

In another embodiment, rather than re-executing the instruction with respect to the ST(i) register (at block 260 of Figure 4), the computer system 10 instead checks the ST(i) register thereby eliminating the need to re-execute an instruction. In this embodiment, additional hardware or microcode is added to determine which register ST(i) is referenced in the instruction.

Accordingly, it is intended that the scope of the invention be only limited as necessitated by the accompanying claims.